title: Using Events

Finding pending transactions with MEV-Share Event Stream

Now that we have all the setup done, we need to find some transactions to backrun. Remember, we're looking for transactions that affect the price of our trading pair (WETH/DAI), so that we can take advantage of the price impact created by the transaction to get a better price for our trade. By placing the two transactions in a bundle (the transaction we found on the event stream, and our backrun trade), we ensure that our trade only executes if it gets placed immediately behind the transaction that causes the price impact that benefits us.

We'll start by listening to the MEV-Share event stream. The event stream shares data about pending transactions (and bundles). We can use this information to deduce whether a transaction affects the price of our trading pair.

In our project, we add a main function at the bottom, and in it, use the mev-share client's`on` function to execute our own code when we receive new pending transaction events. To start, let's just look at the event stream by printing each event to the console.

src/index.ts

```tsx // ... previous code still up here ^

async function main() { console.log("mev-share auth address: " + authSigner.address) console.log("executor address: " + executorWallet.address)

```
// bot only executes one trade, so get the nonce now
const nonce = await executorWallet.getNonce("latest")

mevshare.on('transaction', async (pendingTx: IPendingTransaction) => {
        // callback to handle pending transaction
    console.log(pendingTx)
})
```

} main() ```

Try running this:

`bash npx ts-node src/index.ts`

You should see events popping up in the console. Something like this:

> Traffic is often low on goerli, so you may want to try connecting to mainnet with `MevShareClient.useEthereumMainnet(authSigner)` to see more events.

The logs filling up your console are all transactions which we can backrun. You'll notice that some share more data than others. We'll cover how to include these transactions in our bundles soon, but first we need to go a little deeper to understand these events, and how we might use them to our advantage.

## Finding backrun opportunities using event data

Transactions on MEV-Share can share a wide variety of data with searchers. They can choose to only share their transaction hash to maintain the most privacy, or they can share logs, calldata, the function selector, and/or the `to` address of their transaction. Sharing more data gives searchers more options for running MEV strategies with those transactions, and so improves the chances of those transactions landing on chain quickly.

:::info For the Adventurous

Bundles can also be shared on MEV-Share. If you query the raw stream (you can view it in your web browser here: https://mev-share.flashbots.net), you'll see that each event actually has a `txs` property, which itself may contains transaction events. In the client library, we convert the events into transaction or bundle types for you. To listen for bundles with the client lib, call the function `.on("bundle", ...)`. We'll talk more about this in a later guide.

:::info

In our project, we want to know whether a transaction interacts with the token pair that we want to trade on (in our case, WETH/DAI). We want to know this because these transactions might move the price towards our target price.

To find out which trading pair a transaction is interacting with, we need to look at one of two fields`to` and `logs`, depending on which is shared by the sender of the transaction.

The `to` address is the actual recipient of the transaction. Typically, this would be a router contract like the[Uniswap Universal
```

Router, which allows users of the Uniswap web app to split trades between multiple Uniswap liquidity pools. However, we're not interested in any Uniswap routers. We're looking for the token pair contract (e.g. WETH/DAI). If the to field is the pair address, the transaction sender is trading directly on the pair, which may mean that the transaction is from another searcher, as web UI users would most likely be interacting with the router contract. Nevertheless, if someone is trading on the token pair, we want to try to backrun it.

If the logs field is specified, we look for the pair address in the address field of one of the logs. MEV-Share, by default, shares the pair address in logs for swaps on Uniswap V2/V3 (V4 coming soon), Curve and Balancer. We'll look more in depth at decoding logs in a later guide, but for now it's sufficient to simply look for the pair address — if a log contains this address, then we can be confident that the transaction in the event is a good candidate to backrun.

Add the following code in your project:

src/index.ts

```tsx // preceding code omitted for brevity

function transactionIsRelatedToPair(pendingTx: IPendingTransaction, PAIR_ADDRESS: string) { return pendingTx.to === PAIR_ADDRESS || ((pendingTx.logs || []).some(log => log.address === PAIR_ADDRESS)) } ```

Additionally, we'll need to approve the router to spend our WETH. Add this function:

tsx async function approveTokenToRouter( tokenAddress: string, routerAddress: string ) { const tokenContract = new Contract(tokenAddress, ERC20_ABI, executorWallet) const allowance = await tokenContract.allowance(executorWallet.address, routerAddress) const balance = await tokenContract.balanceOf(executorWallet.address) if (balance == 0n) { console.error("No token balance for " + tokenAddress) process.exit(1) } if (allowance >= balance) { console.log("Token already approved") return } await tokenContract.approve(routerAddress, 2n**256n - 1n) }

Then at the start of your main function, find the smart contract address for the token pair we want to trade, and call our function to approve the router to trade our WETH:

tsx const PAIR_ADDRESS = (await uniswapFactoryContract.getPair( SELL_TOKEN_ADDRESS, BUY_TOKEN_ADDRESS )).toLowerCase() await approveTokenToRouter(SELL_TOKEN_ADDRESS, UNISWAP_V2_ADDRESS)

Then, where we handle new pending transactions, call transactionIsRelatedToPair to see if we should backrun the transaction.

Your main function should similar to this when you're done:

```tsx async function main() { console.log('mev-share auth address: ' + authSigner.address) console.log('executor address: ' + executorWallet.address) const PAIR_ADDRESS = (await uniswapFactoryContract.getPair(SELL_TOKEN_ADDRESS, BUY_TOKEN_ADDRESS)).toLowerCase() await approveTokenToRouter(SELL_TOKEN_ADDRESS, UNISWAP_V2_ADDRESS) const nonce = await executorWallet.getNonce('latest')

```
mevshare.on('transaction', async (pendingTx: IPendingTransaction) => {
    if (!transactionIsRelatedToPair(pendingTx, PAIR_ADDRESS)) {
        console.log('skipping tx: ' + pendingTx.hash)
        return
    }
    console.log(`It's a match: ${ pendingTx.hash }`)
})
```

} ```

If you run the code now, you'll probably see a lot of skipped transactions, but eventually you'll find a match! If you're not seeing any activity, try switching to mainnet:

tsx // const mevshare = MevShareClient.useEthereumGoerli(authSigner) // if you want to connect to mainnet instead: const mevshare = MevShareClient.useEthereumMainnet(authSigner)

:::info For the Adventurous Try logging pendingTx in its entirety; look at all the fields. Or in code, check out the interface directly. See if you can find any patterns in the logs parameter.

:::info